Parallelizing Classical Planning Critical Path Heuristics on the GPU

Hypergraph convolutions for computing h^m

Markus Fritzsche David Speck Daniel Gnad Simon Ståhlberg

ICAPS 2026

Outline

- Motivation
- Critical Path Heuristics
- Hypergraph Convolution
- Engineering
- Experiments
- Takeaways

Classical planning still leaves hardware idle

Observation

Modern hardware is massively parallel, but major classical planners still run heuristic search on a single CPU core.

- Parallelizing the open and closed lists is hard.
- Many heuristics are too cheap for GPU offloading.
- More informative heuristics can amortize parallel overhead.

Candidate

Critical path heuristics h^m are admissible, informative, and built from many independent updates.

Bottleneck

The table has one entry for every atom set of size at most m. This makes h^2 useful but expensive, and h^3 quickly memory-heavy.

Contribution

- Represent the h^m dynamic program as a fixed directed hypergraph.
- Compute one Bellman-Ford-style iteration as a hypergraph convolution.
- Map the update to PyTorch/LibTorch tensor primitives on the GPU.
- Reuse the same graph for batching and multiple cost functions.
- Integrate into Fast Downward and evaluate h^2 on IPC benchmarks.

The h^m recurrence

Let s be a set of atoms obtained by regression.

$$h^m(s) = \begin{cases} 0 & \text{if } s \subseteq I, \\ \min_{(s,a,s') \in R(P)} h^m(s') + \mathrm{cost}(a) & \text{if } |s| \leq m, \\ \max_{s' \in \mathcal{P}_{\leq m}(s)} h^m(s') & \text{otherwise.} \end{cases}$$

Three cases

1. Already true in the initial state: cost 0.
2. Small enough set: regress through one action.
3. Too large: keep the hardest subset of size at most m.

For search state s

Initialize table entries with 0 for vertices $v \subseteq s$ and ∞ otherwise.

h^2 computation example

Let $m = 2$ and consider the subgoal set

$$t = \{p, q, r\}.$$

Since $|t| > 2$, the decomposition case gives

$$h^2(t) = \max_{u \in \mathcal{P}_{\leq m}(t)} h^2(u).$$

The relevant non-trivial subsets are:

$$\{p, q\}, \quad \{p, r\}, \quad \{q, r\}.$$

Suppose action a has

$$\begin{aligned} \mathrm{pre}(a) &= \{x, y\}, \\ \mathrm{add}(a) &= \{p, q, r\}, \\ \mathrm{del}(a) &= \emptyset. \end{aligned}$$

Regressing any pair over a yields $\{x, y\}$:

$$h^2(\{p, q\}) \leq h^2(\{x, y\}) + \mathrm{cost}(a),$$

and analogously for $\{p, r\}$ and $\{q, r\}$.

What the table computes

Every small atom set receives the cheapest cost found by all such regressions.

One h^2 update loop

`stack(A,B)`

pre: H_A, C_B add: O_{AB}, C_A, E

$H_A = \text{holding}(A)$,
$C_X = \text{clear}(X)$
$O_{XY} = \text{on}(X, Y)$,
$E = \text{handempty}$.

Max+add kernel

One CUDA item per edge:

$$u_e = \max_{x \in T_e} d(x) + \text{cost}(e)$$

tail tuple	initial cost
$\emptyset$	0
$\{H_A\}$	0
$\{C_B\}$	0
$\{H_A, C_B\}$	0

$\max(0, 0, 0, 0) + 1 = 1$

Grouped min-update loop

Only generated heads can decrease; repeat while `changed_flag`:

$$d(h) \leftarrow \min(d(h), \min_{e \to h} u_e)$$

tuple	old	after 1
$\{H_A\}$	0	0
$\{C_B\}$	0	0
$\{O_{AB}\}$	∞	1
$\{C_A\}$	∞	1
$\{E\}$	∞	1
$\{H_B\}$	∞	∞
$\{O_{BA}\}$	∞	∞
$\{O_{AB}, C_A\}$	∞	1
$\{O_{AB}, E\}$	∞	1
$\{C_A, E\}$	∞	1
...	...	...
$\{O_{AB}, O_{BA}\}$	∞	∞

Hypergraph representation

Vertices

$$X = \mathcal{P}_{\leq m}(V)$$

Every vertex is one dynamic-programming table entry.

Hyperedges

Each regression transition (s, a, s') induces edges

$$T = \mathcal{P}_{\leq m}(s'), \qquad H = \{v\}, \quad v \in \mathcal{P}_{\leq m}(s),$$

with weight $\mathrm{cost}(a)$.

Important separation

The hypergraph depends on the planning task and m, not on the current search state.

- Search states only initialize vertex labels.
- The same graph can be reused for many heuristic evaluations.
- This makes batching natural.

One convolution round

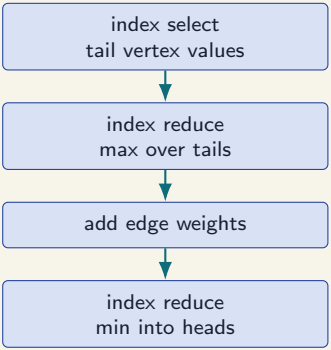
For each hyperedge $e = (T, \{v\})$:

$$u_e^t = \max_{x \in T} d^t(x) + \mathrm{weight}(e).$$

Then update each head vertex by the cheapest incoming proposal:

$$d^{t+1}(v) = \min\left(d^t(v), \min_{e=(T,\{v\}) \in E} u_e^t\right).$$

Repeat until the vertex labels reach a fixed point.



The paper implementation uses PyTorch primitives with exactly this select/reduce/reduce structure.

Tensor implementation from the paper

```
all_vals = index_select(
    input=vertices, dim=0,
    index=value_indices)
max_vals = index_reduce(
    input=all_vals, dim=0,
    index=max_indices, source=all_vals,
    reduce="amax", include_self=False)
max_vals = max_vals + edge_weights
vertices = index_reduce(
    input=vertices, dim=0,
    index=min_indices, source=max_vals,
    reduce="amin", include_self=True)
```

Select tail values, reduce by max, add edge weights, then reduce by min into head vertices.

Computing a heuristic value

Given a search state s:

$$d^0(v) = \begin{cases} 0 & \text{if } v \subseteq s, \\ \infty & \text{otherwise.} \end{cases}$$

Let d^* be the converged labeling. The heuristic is

$$h^m(s) = \max_{g \in \mathcal{P}_{\leq m}(G)} d^*(g).$$

Why this is correct

The hypergraph is the h^m hypergraph of Steinmetz and Torralba: at convergence, $d^*(v) = h^m(v)$ for every vertex v.

GPU-friendly structure

The entire hypergraph is processed in parallel each round, then the same operations repeat until convergence.

Implementation strategy

- Integrated into Fast Downward.
- Hypergraph construction runs once on the CPU.
- Vertex arrays, edge indices, and weights are transferred to GPU.
- LibTorch 2.1.2 and CUDA 12.1 execute the convolution rounds.

Avoiding synchronization

Checking convergence every round forces CPU-GPU synchronization. The implementation tracks the maximum observed number of iterations and delays checks until that threshold.

CUDA graphs

Repeated GPU kernel launches are captured to reduce launch overhead.

How many GPU threads?

CUDA launch shape

The fused implementation uses fixed-size blocks:

$$\texttt{threads} = 256, \qquad \texttt{blocks} = \left\lceil \frac{\texttt{work}}{256} \right\rceil .$$

Logical work

max+add: $n_{\mathrm{cp}} \cdot B \cdot |E|$
grouped min: $n_{\mathrm{cp}} \cdot B \cdot |H_{\mathrm{grp}}|$

Example: $n_{\mathrm{cp}} = 2$, $B = 3$, $|E| = 5$
$2 \cdot 3 \cdot 5 = 30$ edge work items

drawn with toy 8-thread blocks;
implementation uses 256-thread blocks.

What changes the thread count?

More cost partitions, larger batches, and more hyperedges/head groups create more logical work. Tail size affects work per thread.

Batching and cost functions

Batching

Compute h^2 for all successors of an expanded state in parallel.

- Same hypergraph structure.
- Multiple vertex-label columns.
- Better use of GPU throughput.

Cost partitioning

Use cost functions $c_1, \ldots, c_n$ satisfying

$$c_1(a) + \cdots + c_n(a) \leq \mathrm{cost}(a).$$

- The graph topology is shared.
- Edge weights differ per cost function.
- The additive heuristic remains admissible.

Dominance pruning

Theorem

Let $e_1 = (T_1, H)$ *and* $e_2 = (T_2, H)$ *be hyperedges with the same head. If* $T_1 \subseteq T_2$ *and* $\mathrm{weight}(e_1) \leq \mathrm{weight}(e_2)$*, then removing* e_2 *does not change the computed vertex values.*

Reason

A larger tail can only increase the max-reduction. A larger edge weight cannot compensate for that.

Cost partitioning caveat

With several cost functions, dominance must hold for every cost function, so the pruning condition becomes stricter.

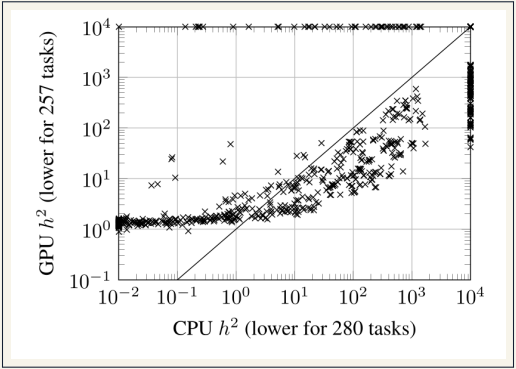
Experimental setup

- Benchmarks from the optimal tracks of IPC.
- Planner configurations use A* search.
- Limits: 30 minutes and 8 GiB CPU memory.
- GPU: one NVIDIA A100 with 40 GB.

Compared configurations

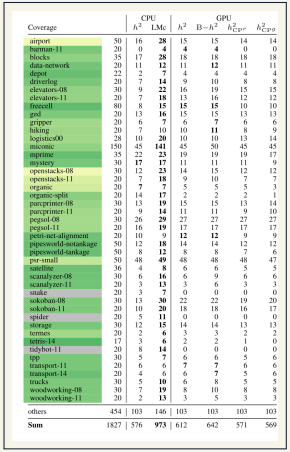
- CPU h^2 generalized Bellman-Ford baseline.
- GPU h^2 convolution.
- Batched GPU B-h^2.
- Goal and random cost partitioning variants.
- LM-cut as a strong h^1-based baseline.

Runtime: GPU pays off on large instances



The GPU has startup and construction overhead, but large instances see speedups of up to two orders of magnitude.

Coverage



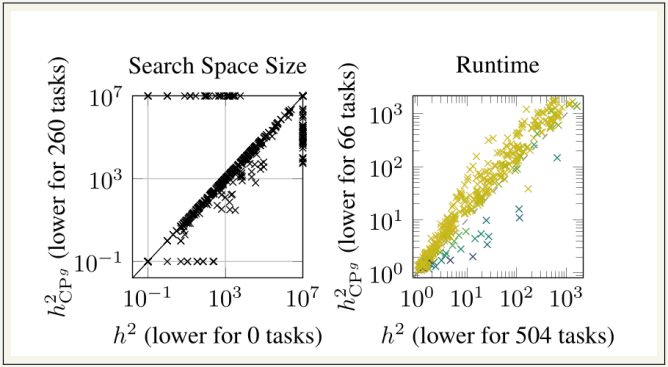
Coverage	CPU		GPU				
	h^2	LMc	h^2	B−h^2	h^2_{CPU}	h^2_{CPU}	h^2_{CPU}
airport	50	16	**28**	15	15	14	14
barman-11	20	0	**4**	**4**	**4**	0	0
blocks	35	17	**28**	18	18	18	18
data-network	20	11	**12**	11	**12**	11	11
depot	22	2	7	4	4	4	4
driverlog	20	7	**14**	9	10	8	8
elevators-08	30	9	**22**	16	19	15	15
elevators-11	20	7	**18**	13	16	12	12
freecell	80	8	**15**	**15**	**15**	10	10
grid	20	13	**16**	15	15	13	13
gripper	20	6	**7**	6	**7**	6	6
hiking	20	7	10	10	**11**	8	9
logistics00	28	10	**20**	10	10	13	14
miconic	150	45	**141**	45	50	45	45
mprime	35	22	**23**	19	19	19	17
mystery	30	**17**	**17**	11	11	11	9
openstacks-08	30	12	**23**	14	15	12	12
openstacks-11	20	7	**18**	9	10	7	7
organic	20	**7**	**7**	5	5	5	3
organic-split	20	14	**17**	2	2	2	1
parcprinter-08	30	13	**19**	15	15	13	14
parcprinter-11	20	9	**14**	11	11	9	10
pegsol-08	30	26	**29**	27	27	27	27
pegsol-11	20	16	**19**	17	17	17	17
petri-net-alignment	20	10	9	**12**	**12**	9	9
pipesworld-notankage	50	12	**18**	14	14	12	12
pipesworld-tankage	50	8	**12**	8	8	7	6
psr-small	50	48	**49**	48	48	48	47
satellite	36	4	**8**	6	6	5	5
scanalyzer-08	30	6	**16**	6	9	6	6
scanalyzer-11	20	3	**13**	3	6	3	3
snake	20	3	7	0	0	0	0
sokoban-08	30	13	**30**	22	22	19	20
sokoban-11	20	10	**20**	18	18	16	17
spider	20	5	**11**	0	0	0	0
storage	30	12	**15**	14	14	13	13
termes	20	2	**6**	3	3	2	2
tetris-14	17	3	**6**	2	2	1	0
tidybot-11	20	8	**14**	0	0	0	0
tpp	30	5	**7**	6	6	5	6
transport-11	20	6	6	**7**	**7**	6	6
transport-14	20	4	6	6	**7**	5	6
trucks	30	5	**10**	6	8	5	5
woodworking-08	30	7	**19**	8	10	8	8
woodworking-11	20	2	**13**	3	5	3	3
others	454	103	146	103	103	103	103
Sum	1827	576	**973**	612	642	571	569

What to read

Per-domain coverage on IPC optimal-track benchmarks.

- GPU batching solves the most instances among the h^2 variants.
- GPU h^2 improves over CPU h^2 in total coverage.
- LM-cut remains stronger overall, but parallel h^2 wins in some domains.

Cost partitioning: stronger heuristic, too expensive



Goal-based cost partitioning reduces some search spaces, but runtime and memory overhead dominate on many instances.

What we learned

- h^m can be expressed as repeated convolutions on a fixed hypergraph.
- That formulation maps directly to GPU tensor operations.
- GPU h^2 substantially improves runtime and coverage over CPU h^2.
- Batching is important: it better matches the GPU execution model.
- LM-cut is still stronger overall, but GPU heuristic evaluation is a practical entry point for parallel planning.

Conclusion

- Classical planning can use GPUs without parallelizing the entire search algorithm.
- The useful unit of parallelism is heuristic evaluation.
- Hypergraph convolution gives a clean bridge from h^m theory to GPU implementation.

Code

`downward-h2`

Thank you.

Questions & discussion

markus.fritzsche@liu.se